MCU Machine Learning With Edge Impulse

An intuitive SDK for tinyML

By Andrea Garrapa

Interest in machine learning has exploded recently, and there are many cloud-based solutions that place its power in our hands. However, in applications with limited connectivity, operating on sensitive data, or demanding low latency, it would be preferable to execute such capability on small microcontrollers. Using a simple example project, we examine the Edge Impulse software development kit to see how it supports developers in deploying machine learning capability on tiny, low-power, and economical microcontrollers.

The use of machine learning (ML) algorithms [1] at the point of application, such as plant in an industrial complex, is known as edge computing. This manner of deploying ML in Industry 4.0 and Internet of Things (IoT) applications ensures sensitive data stays on-site and allows ML-based decisions to be utilized immediately with low latency. Training the models requires a lot of computing power, but executing them can be undertaken on small, cost-effective microcontrollers. This asymmetric approach to ML is being studied intensely and has led to the establishment of the tinyML Foundation [2]. There are three main reasons for running machine learning models on embedded devices:

- **Local processing**: avoids transmission of data collected by the sensors. There is no need for an internet connection resulting in fewer deployment limitations.
- **Low power consumption**: microcontrollers use very little energy. A microcontroller can run an image recognition algorithm continuously for a year on a single coin cell battery.
- **Cost and deployment**: microcontrollers are inexpensive and ubiquitous, making them perfect for deploying ML models. New or improved models can be rolled out with a simple software update, avoiding replacing hardware.

While tinyML is a relatively new research field, the applications to which it can be applied are innumerable, ranging from agriculture and medicine to predictive maintenance.

Edge Impulse

Edge Impulse, also known as tinyML as a service, delivers machine learning to developers via an open-source device software development kit (SDK). It enables simple data collection from sensors, real-time signal processing, testing, and deployment to any target device. The open-source SDK allows data to be collected from, or deployed to, any device. Users can contribute and extend both the algorithms and support for target devices. The device software, including SDK, client, and generated code, is provided as open-source with an Apache 2.0 license. Collaboration with the TensorFlow Lite Micro project enables support for the broadest range of ML architectures, operators, and targets. The SDK is free for individual developers, while an enterprise version is available by subscription for teams implementing tinyML in innovative products.

Intuitive and straightforward to use, you're ready to start your first tinyML project after creating and logging into your account [3]. The SDK interface is clean and uncluttered (**Figure 1**).

Once logged in, the *Dashboard* provides an overview of our projects and information about them. A guide describing how to start your first project, or continue an existing one, is also available. Further options are listed under *Dashboard* on the

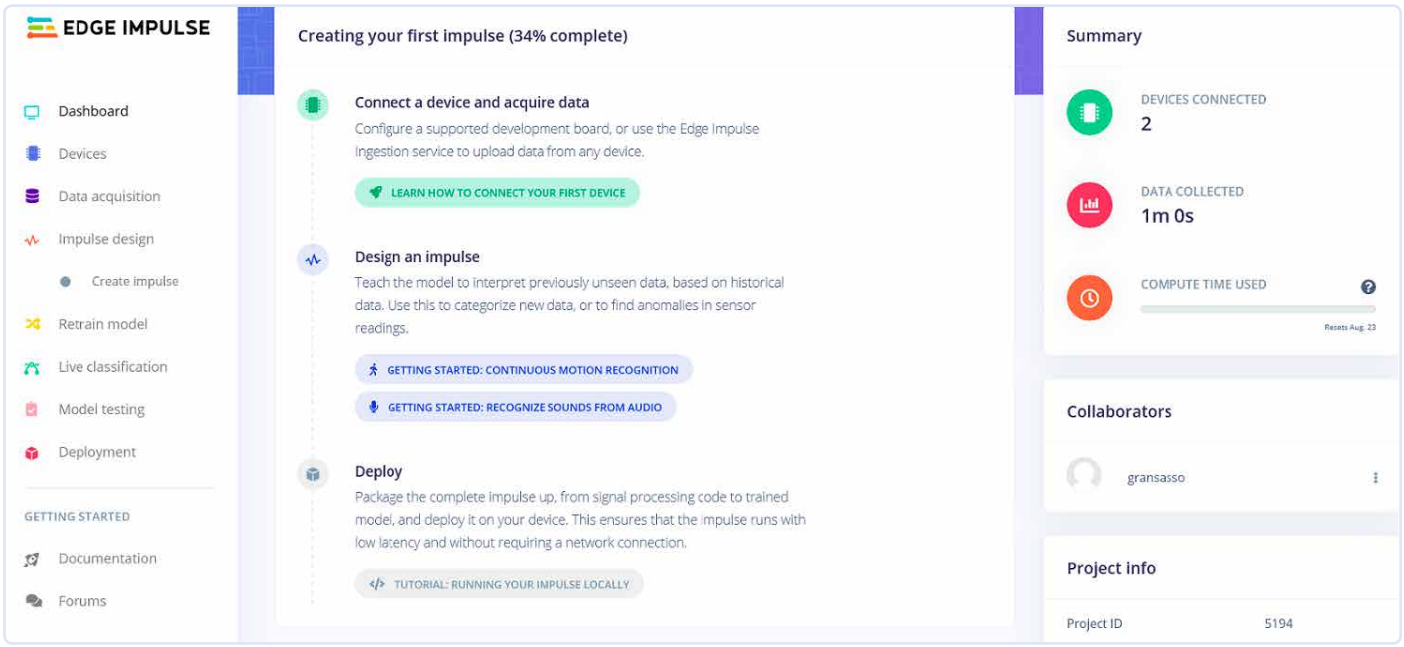

Figure 1: Edge Impulse SDK dashboard.

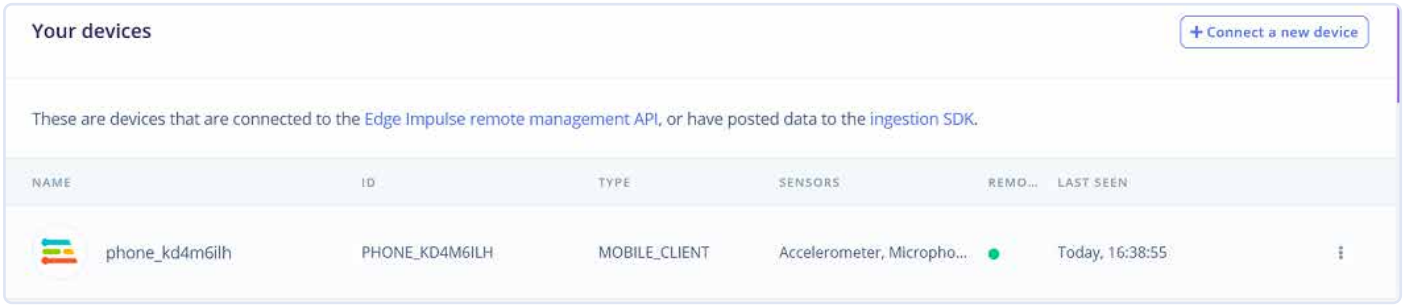

Figure 2: The Devices tab showing the available connected devices.

left-hand side. The order of these options reflect the development stages needed to realize a project. For example, the *Device* tab lists connected devices and allows the connection of new ones. *Data acquisition* displays data collected for testing and training. *Impulse design* supports creating an impulse, a procedure that accepts raw data, uses signal processing to extract features, then classifies the new data using a learning block.

To connect an embedded device, the directions in the middle section of the *Dashboard* should be followed. This process can also be implemented using the *Devices* tab and clicking on *Connect a new device*. The devices supported by the development environment are as follows:

- **ST IoT Discovery Kit:** also known as B-L475E-IOT01A, this development board features a microcontroller using a Cortex-M4 processor, MEMS motion sensors, a microphone, and WiFi.
- **Arduino Nano 33 BLESense:** a tiny development board with a Cortex-M4-based microcontroller, motion sensors, a microphone, and BLE.
- **AI ECM3532 Eta Compute Sensor:** a tiny development board featuring the TENSAI ECM3532 SoC with a Cortex-M3-based microcontroller and a separate CoolFlux DSP to accelerate machine learning operations. The board includes two microphones, a 6-axis accelerometer/gyroscope, and a pressure/temperature sensor. The ECM3532 SoC on the development board supports Continuous Voltage Frequency Scaling. This allows clock frequency and voltage to be scaled at runtime to maximize power efficiency to achieve ultra-low power machine learning algorithms.
- **Smartphone:** as long as a modern browser is available, smartphone clients are also fully supported by Edge Impulse. The phone behaves like any other device and any data and models created can be deployed to embedded devices.

The devices listed can all be used to sample raw data, build models, and deploy trained machine learning models. After connecting one or more devices, they are listed in the *Devices* table (**Figure 2**). With one or more devices connected to Edge Impulse, it is now possible to teach a model to interpret data it has never seen before, based on historical data. It can then be used to classify new data or find anomalies in sensor readings.

Two tutorials are offered in the guide on the *Dashboard*: one for audio and another for gesture recognition. Here we will review the audio tutorial.

Sound recognition

In this tutorial, machine learning is used to create a system that can recognize when a particular sound occurs, an activity known as audio classification. The system developed will be able to identify the sound of water running from a faucet, even in the presence of other background noise. The tutorial teaches how to collect audio data from microphones, use signal processing

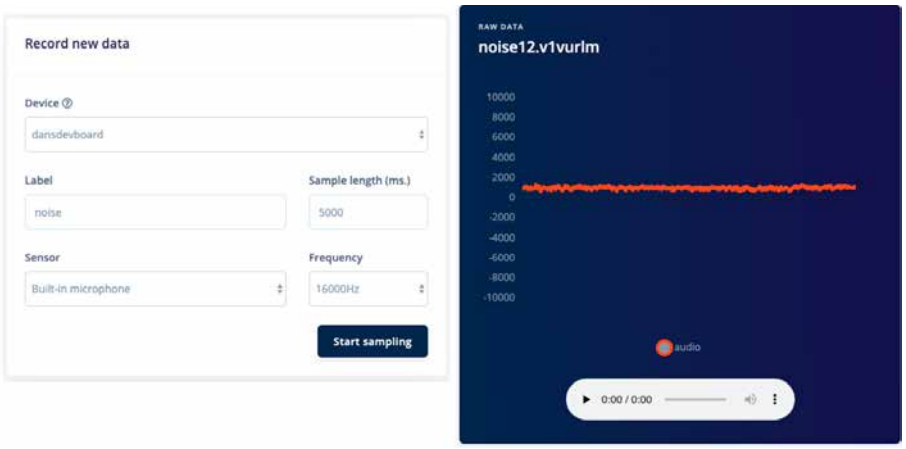

Figure 3: Parts of the Data acquisition tab for recording and playback of audio data.

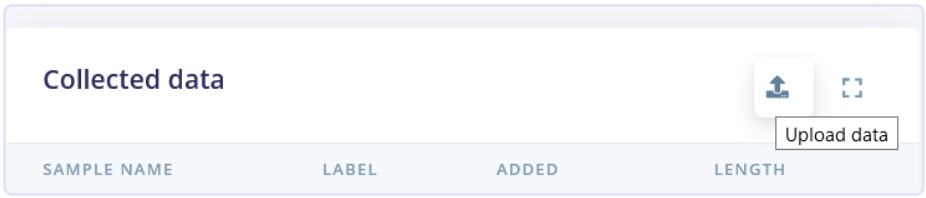

Figure 4: Data acquisition tab used to upload predefined datasets.

to extract the most important information, and train a deep neural network to determine if running water can be heard in a given audio clip. Finally, the model will be deployed on an embedded device to evaluate its operation.

Collect data

First, we need to collect some audio data that will be used to train the machine learning model. The goal is to detect the sound of water flowing from an open faucet. This requires collection of some examples of this sound. Additionally, examples of typical background noises that do not contain the sound of a faucet are needed also. This enables the model to learn to discriminate between the two. These two types of example sounds represent the two classes in our model: *background noise* or *faucet operation*. The sensor data can be collected by your device using the *Data acquisition* tab in the SDK. This is where all the raw data is stored and, if the device is connected to the Remote Management API, where you can start sampling new data.

Let's start by recording a sample of background noise that does not contain the sound of an open faucet. In the *Record new data* section select the device, set the label to noise, define a sample length of 1000, and choose the built-in microphone as the sensor. This defines that you want to record one second of audio and label the recorded data as noise. The labels can be edited later if required. After clicking *Start sampling*, the device will capture one second of audio and transmit it to Edge Impulse. Once the data is loaded, a new row will appear under *Collected data*. In the *Raw Data* section, you can analyze the recorded signal waveform and listen to the audio using the controls provided (**Figure 3**).

Building a dataset

From here, we can now create a dataset. For a simple audio classification model like this, we should aim to capture around 10 minutes of data. With two classes defined, our data should ideally be equally balanced between each. So we'll collect:

- Five minutes of background noise, labeled "noise".
- Five minutes sound of the faucet open, labeled "faucet".

In the real world, there are usually additional sounds superimposed on the intentional sounds. For example, an open faucet is often accompanied by the sound of dishes being washed or a conversation in the kitchen. Background noise might also include the sound of a television, children playing, or cars driving by. The training data must contain these real-world sounds. If the model is not exposed to these sounds

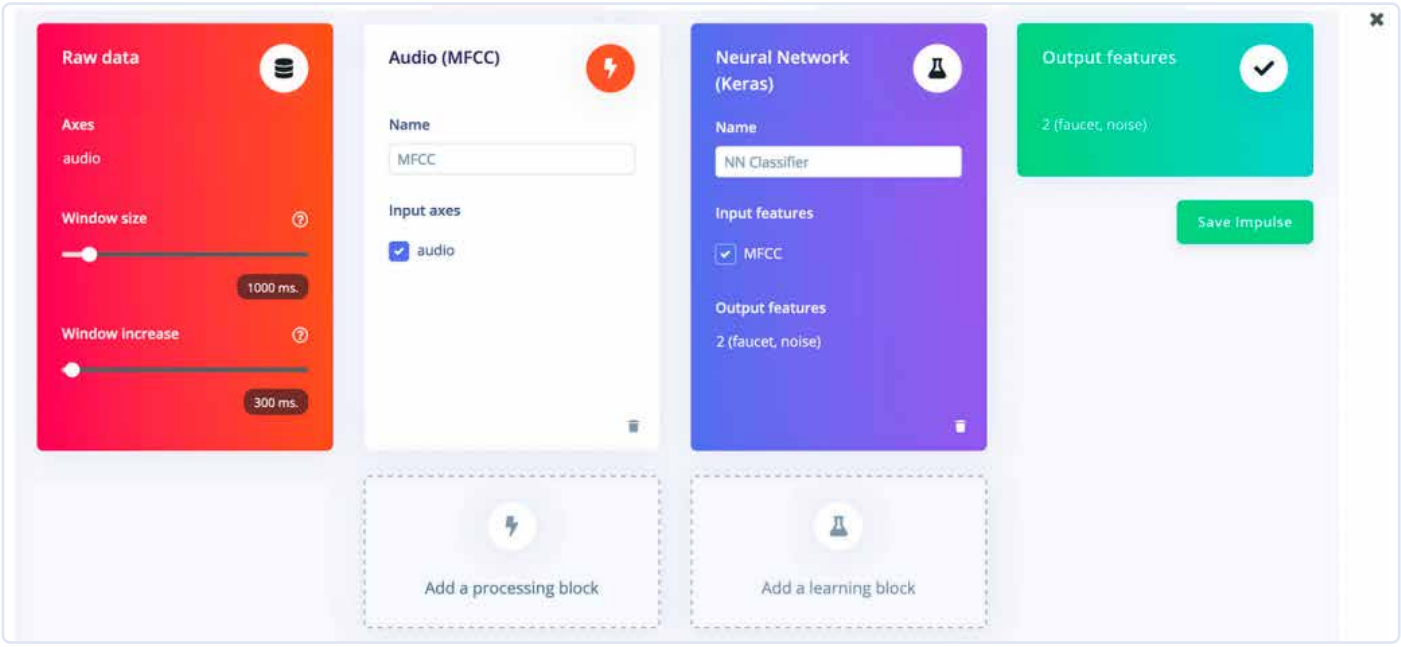

Figure 5: Impulse creation.

during training, it will not learn to take them into account and will not perform as well. For this tutorial, we will capture the following:

- Background noise:
  - Two minutes of background noise without much additional activity.
  - One minute of background noise with TV or music playback.
  - One minute of background noise with occasional speech or conversation.
  - One minute of background noise with the sounds of household chores.
- Open faucet noise:
  - One minute of an open faucet.
  - One minute of a different open faucet.
  - One minute of an open faucet with a TV or music playing.
  - One minute of open faucet with casual chatting or conversation.
  - One minute of a running faucet with sounds of household chores.

If it is not possible to acquire this diversity of audio samples, there is no need to worry. The main aim here is to have five minutes of real-world data for each class as the resultant model will work better if it has been trained on a representative dataset. There is also no guarantee that the model will operate well in the presence of sounds that were not included in the training set. This is why it is essential to make the dataset as varied and representative of real-world conditions as possible. Alternatively, pre-built datasets can be downloaded containing around ten minutes of data of the required classes [4]. After downloading the dataset, unzip the file to a location of your choice and, on the data capture page, click on the *Upload data* icon visible in **Figure 4.** Select the files and label them directly here.

The amount of audio that can be captured in one shot varies depending on the hardware's memory. The ST B-L475E-IOT01A board has enough memory to capture 60 seconds of audio at a time, while the Arduino Nano 33 BLE Sense has enough memory for 16 seconds. To capture 60 seconds of audio, the sample length must be set to 60,000. Since the board transmits data rather slowly, it will take around 7 minutes before a 60-second sample is displayed in Edge Impulse. After acquiring the 10 minutes of data needed, it is time to start designing an impulse.

Design an impulse

The impulse takes the raw data, breaks it down into smaller windows, then uses:

- Signal processing blocks to extract features.
- A learning block to classify new data.

Signal processing blocks always return the same values for the same input and are used to simplify the processing of raw data, while learning blocks learn from past experience. For this tutorial, we will use the signal processing block *MFCC*. MFCC stands for Mel-Frequency Cepstral Coefficients [5]. It sounds scary, but basically it's just a way to transform unprocessed audio that contains a large amount of redundant information into a simplified form. We will then pass this simplified audio data into a neural network block (learning block) that will learn to distinguish between the two audio classes (faucet and noise). This is configured in the *Create impulse* tab. The first thing you see upon opening this section is *Raw Data*.

As mentioned earlier, Edge Impulse splits the raw samples into windows that feed into the machine learning model during training. The *Window size* field controls the length, in milliseconds, of each data window. A one-second audio sample will be enough to determine if a faucet is open or not, so you need to make sure the window size is set to 1000 ms. Each raw sample is split into multiple windows, and the *Window increase* field controls the offset of each subsequent window from the first. For example, a window increase value of 1000 ms would cause each window to start one second after the start of the previous one.

By setting *Window increase* lower than the window size, we can create overlapping windows. This is actually a very good idea. Although they may contain similar data, each overlapping window is still a unique example of audio representing the sample label. Using overlapping windows allows us to make the most of the training data. For example, with a window size of 1000 ms and a window increment of 100 ms we can extract ten unique windows from only two seconds of data. For this example, we will use a 1000 ms long window and a window increase field set to 300 ms. Clicking on *Add a processing block* allows us to select

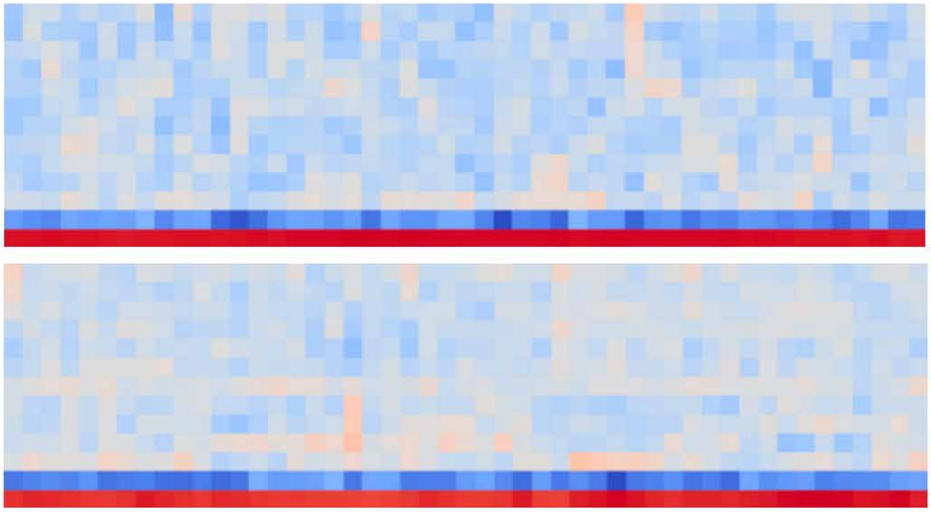

Figure 6: Comparison of the spectrograms of noise (top) and faucet sound (bottom).

the *MFCC* block while clicking *Add a learning block* allows selection of the *Neural Network (Keras)* block. To finish, click on *Save impulse*. The created impulse should look as shown in **Figure 5**.

MFCC block configuration

Now that we have assembled the building blocks of our impulse we can configure each individual part. Clicking on the *MFCC* tab in the navigation menu on the left takes you to the block configuration page to preview how the data will be

Figure 7: Post-training network performance.

Figure 8: Results of testing data classification.

transformed. The page's right side shows a visualization of the MFCC output for an audio track, known as a spectrogram. The MFCC block transforms an audio window into a table of data where each row represents a frequency range, and each column represents a time period. The value contained in each cell reflects the magnitude of its associated frequency range during that time frame. The spectrogram shows each cell as a colored block with the change in color intensity describing the amplitude. **Figure 6** visually compares the spectrograms of noise (top) and faucet sound (bottom).

The differences are difficult for a person to see, but they are different enough for a neural network to learn how to classify them. There are many different ways to configure the MFCC block in the *Parameters* box. Edge Impulse provides reasonable defaults that will work well for many use cases, so we can leave these values unchanged. The spectrograms generated by the MFCC block will be passed to a neural network that is particularly good at learning to recognize patterns in this type of tabular data.

Before training our neural network, we will need to generate MFCC blocks for all of our audio windows. To do this, we need to click on the *Generate feature* button at the top of the page, then click on the green *Generate feature* button. With ten minutes of training data to analyze, the process will take a few minutes to complete. Once this process is complete, the *Features explorer* will show a visualization of the dataset. Here dimensionality reduction is used to map features to a 3D space, and you can review if the different classes separate well or find mislabeled data.

Neural network configuration

Neural networks are algorithms loosely modeled on the human brain to learn how to recognize the patterns in their training data. The network we are training will take the MFCC as an input and try to map it to one of two classes: noise or faucet. Clicking on *NN Classifier* on the left menu will open the block configuration window. A neural network is composed of layers of virtual neurons, as represented on the page's left side. An input, in our case an MFCC spectrogram, is fed into the first layer of neurons that filters and transforms it based on the unique internal state of each neuron. The first layer's output is then fed into the second layer and so on, gradually transforming the original input into something radically different. In this case, the spectrogram input is transformed over four intermediate layers into just two numbers: the probability that the input represents noise and the probability that the input represents a running faucet.

During training, the neurons' internal state is gradually optimized and refined so that the network transforms its input in the correct way to produce the desired output. This is done by entering a sample of training data, checking the distance between the network's output and the correct response (its label), and adjusting the neurons' internal state to make it more likely to produce the correct response next time. When done thousands of times, this results in a trained network.

A particular arrangement of layers is called an architecture, and different architectures are useful for different tasks. The default neural network architecture provided by Edge Impulse works well for the current project, but you can also define your own architectures. Before you start training, you need to change some values in the configuration. First, set the *Number of training cycles* to 300. This means that the entire data set will be run 300 times through the neural network during training. If too few cycles are run, the network will not learn all it can from the training data. However, if too many cycles are run, the network may overfit the training data and will no longer work well on data never seen before. This problem is called overfitting.

Next, you need to change *Minimum confidence rating* to 0.7. This means that when the neural network makes a prediction (for example, the probability that some audio contains a running faucet is 0.8), Edge Impulse will ignore it unless it is above the 0.7 threshold. To begin training, click *Start training*. The training will take a few minutes. When finished, you will see the *Last training performance* panel at the bottom of the page (**Figure 7**).

The neural network in Edge Impulse is now fully trained, but what do all those numbers

mean? At the beginning of training, 20% of the data is set aside for validation. This means that, instead of being used to train the model, it is used to evaluate model performance. The panel *Last training performance* shows the results of this validation, providing some information about the model and how it works. The numbers may differ from those in this tutorial. The *Accuracy* parameter refers to the percentage of audio windows that were correctly classified. The higher the number, the better, although accuracy approaching 100% is unlikely and is often a sign of overfitting. For many applications, an accuracy greater than 80% can be considered very good. The confusion matrix [6] is a table that shows the balance between correctly and incorrectly classified windows.

Classify new data

The previous step's performance numbers show that the model is performing well on its training data, but it is vital to test the model on new data before deploying it. This will help us ensure that the model has not overfitted on the training data. Edge Impulse provides some useful tools for testing the model, including capturing real-time data from the device and immediately attempting to classify it. To test it, simply click on *Live classification* in the menu on the left. The device should appear in the *Classify new data* panel. Clicking *Start sampling* will capture five seconds of background noise.

Test the model

Using the *Live classification* tab you can quickly test the model and get an idea of its behavior. But, to be sure that it works well, you need to do more rigorous testing. This is where the *Model testing* tab comes in. Ideally, you should collect a test set that contains at least 25% of the quantity of data used in the training set. So, for ten minutes of training data you need to collect at least 2 minutes 30 seconds of test data. You must also ensure that this test data represents a wide range of possible conditions, thus evaluating the model's performance with many different types of inputs.

For example, collecting test audio for several faucets is a good idea. You can use the *Data Acquisition* tab to manage your test data. Open the tab, then click *Test data* at the top. From here, use the *Upload* function as was done for the training data but this time upload as testing data. Make sure the samples are labeled correctly. When finished, return to the *Model testing* tab, select all samples, and click *Classify selected*.

Figure 8 shows the results of the classification. The panel shows that the model performs with an accuracy of 73.42%. For each test sample, the panel shows a breakdown of its individual performance. For example, one of the samples was classified with an accuracy of 67%. Samples that contain many misclassifications are valuable since they contain examples of audio types to which the model currently does not fit. These are often worth adding to the training data.

Distribution on a hardware device

With the impulse designed, trained, and verified, deploying the model on a hardware device is now possible. This runs the model without an Internet connection at minimal latency and operates with the lowest power consumption. Edge Impulse can package the entire impulse, including the MFCC algorithm, neural network weights, and classification code, into a single C++ library that you can include in your embedded software. To export the model, click on *Deployment* in the menu. Then, under *Build firmware* select the correct development tab and click *Build*. This will export the impulse and create a binary that will

run on the development board in one step. When the build is complete, you will be asked to download a binary file. Simply save this to your computer. Upon clicking the *Build* button, a pop-up will appear with text and video instructions on how to deploy the binary to the particular device selected. Following these instructions will allow you to test it once you're done. We are able to connect to the firmware we have just flashed to the board via a serial interface by opening a terminal and running:

```
$ edge-impulse-run-impulse
```

This will capture the microphone's audio, run the MFCC code, and then classify the spectrogram.

Conclusions

As a young field of research, ML on microcontrollers is complicated for the uninitiated. The Edge Impulse SDK facilitates this process by simplifying the capture and analysis of data, and training and building neural network models for deployment on microcontrollers. There are endless applications for this type of model, from monitoring industrial machinery to recognizing voice commands. Edge Impulse is easy to use, intuitive, and, being free, means you can start immediately exploring tinyML for embedded devices.

This article was first published in Italian by Elettronica Open Source (https://it.emcelettronica.com). Elektor translated it with permission.

210259-01

WEBLINKS

[1] "What is Machine Learning? A Definition," Expert.ai, May 2020: https://bit.ly/3tfuyQL
[2] tinyML Foundation: https://www.tinyml.org/
[3] Edge Impulse website: https://www.edgeimpulse.com/
[4] Dataset of faucet sounds: https://cdn.edgeimpulse.com/datasets/faucet.zip
[5] "Mel-frequency cepstrum," Wikipedia: https://en.wikipedia.org/wiki/Mel-frequency_cepstrum
[6] "Simple guide to confusion matrix terminology," data school, March 2014: https://bit.ly/32eaoup